

# REACT: A Solidarity-based Elastic Service Resource Reallocation Strategy for Multi-access Edge Computing

Alisson Medeiros<sup>a</sup>, Torsten Braun<sup>a</sup>, Antonio Di Maio<sup>a</sup> and Augusto Neto<sup>b,c</sup>

<sup>a</sup>Institute of Computer Science, University of Bern, Switzerland

<sup>b</sup>Informatics and Applied Mathematics Department, Federal University of Rio Grande do Norte, Brazil

<sup>c</sup>Instituto de Telecomunicações, Portugal

## ARTICLE INFO

### Keywords:

5G  
Multi-access Edge Computing  
Resource elasticity  
Auto-scaling  
Self-adaptive  
Virtualization

## ABSTRACT

The Multi-access Edge Computing (MEC) paradigm promises to enhance network flexibility and scalability through resource virtualization. MEC allows telecom operators to fulfill the stringent and heterogeneous requirements of 5G applications via service deployment at the edge of the mobile network. However, current solutions to support MEC struggle to provide resource elasticity since MEC infrastructures have limited resources. The coexistence of many heterogeneous services on the distributed MEC infrastructure makes the resource scarcity problem even more challenging than it already is in traditional networks. Services need distinct resource provisioning patterns due to their diverse requirements, and we may not assume an extensive MEC infrastructure that can accommodate an arbitrary number of services. To address these aspects, we present REACT: a MEC-supported self-adaptive elAstiCiTy mechanism that leverages resource provisioning among different services running on a shared MEC environment. REACT adopts an adaptive and solidarity-based strategy to redistribute resources from over-provisioned services to under-provisioned services in MEC environments. REACT is an alternative strategy to avoid service migration due to resource scarcity. Real testbed results show that REACT outperforms Kubernetes' elasticity strategy by accomplishing up to 18.88% more elasticity events, reducing service outages by up to 95.1%, reducing elasticity attempts by up to 95.36%, and reducing over-provisioned resources by up to 33.88%, 38.41%, and 73% for CPU cycles, RAM and bandwidth resources, respectively. Finally, REACT reduces response time by up to 15.5%.

## 1. Introduction

The realization of the 5G architecture (including 5G beyond approaches, like 6G or Networking 2030) is guided by novel technologies and new trends in user demands for modern applications, such as tactile Internet, autonomous vehicles, immersive media services, eHealth, etc [1]. To support these new, especially latency-sensitive, applications, services must be deployed at the network edges [2]. A *service* is defined as a virtualized instance of a physical function that is "cloudified" and placed in cloud hosts or network edges, e.g., video transcoding, load balancing, content caching, network address translation, etc [3].

Modern applications demand extreme network and computing performance. Their quality depends significantly on the mobile network infrastructure's elasticity. Resource elasticity is defined as a system's ability to adapt to service workload fluctuations by adjusting resource configurations and provisioning close to the demand [4]. Therefore, elasticity strategies to support stringent and heterogeneous requirements imposed by current and upcoming 5G applications become essential to accelerate their adoption.

Following this trend, telecom operators have adopted the *telco-cloud* paradigm [5] to support on-demand MEC resource elasticity. In this sense, telecom operators are broadly redefining their cloud infrastructures following the MEC concept to achieve the requirements of 5G applications [6, 7].

MEC provides computing resources at the network edges, allowing telecom operators to fulfill latency requirements for future applications and offer service delivery at the edge of the mobile network [8]. One primary problem with MEC is that it has limited computing and communication resources [9, 10]. This may negatively affect quality of service (QoS) in high service demand situations, as network or MEC resources may become insufficient to support them [11]. To maintain satisfactory QoS in these circumstances, services typically migrate from overloaded to less loaded MEC servers [12]. However, this approach requires service check-pointing and restarting for *stateful* services, which may lead to long service downtime if the migration process has to transfer a large amount of data [13, 14].

When resources become scarce, the elasticity mechanism will not meet the ideal resource allocation of the new service load. Hence, the elasticity mechanism triggers, in turn, the time-costly migration procedure, leading to the search for another cloud or edge server to deploy the target service. Although the migration will perhaps succeed in meeting the needed performance at another server, the resulting migration costs might be too high, e.g., downtime and migration time, as the whole migration time is extremely time-consuming [3].

Optimal resource provisioning for MEC is an ongoing challenge [15]. On the other hand, many works in cloud computing propose new resource-elasticity strategies [16, 17]. However, it is essential to develop elasticity strategies adapted to MEC since edge servers may run out of resources as service providers offer more resources for applications as consumer demand increases [18].

\*Corresponding author

✉ alisson.medeiros@inf.unibe.ch (A. Medeiros);

torsten.braun@inf.unibe.ch (T. Braun); antonio.dimaio@inf.unibe.ch (A.D. Maio); augusto@imap.ufrn.br (A. Neto)

State-of-the-art resource elasticity algorithms are reactive, meaning that the auto-scaling procedure is started only after the service's resource usage crosses a predefined threshold. Some of the most popular reactive elasticity solutions, such as Amazon EC2, Microsoft Azure, and Google Cloud Platform (GCP), deploy heuristic auto-scaling schemes, as reactive-based solution meet cloud demands [16].

For the schemes mentioned above, after an elasticity request, the elasticity mechanism will fail to provide auto-scaling procedures when the requested resources are no longer available in a MEC server. As a result, the reactive model is likely to produce multiple attempts until it matches the resource configurations that suit the new service load.

We define the time needed for the auto-scaling procedure to converge and find a suitable resource allocation as *elasticity attempt window*. During the *elasticity attempt window*, the service will suffer from quality degradation due to resource saturation until matching optimal new resource patterns. The situation becomes critical in MEC when resources become scarce, where current reactive models are efficient under resource availability conditions.

We argue that, due to the limited resource characteristics of MEC, its resources must be enhanced to support 5G application deployments through resource elasticity strategies that consider both MEC resource limitations and 5G application requirements. Thus, we assume that over-provisioned resources must exist in virtualized MEC servers that support multi-tenancy, preventing virtual entities, i.e., containers, virtual machines, from being provisioned whenever their load changes. However, it will lead to low-efficiency usage of MEC and increase deployment costs. Based on this, our research focuses on proposing a heuristic elasticity solution tailored to MEC systems, capable of overcoming resource scarcity and resource over-provisioning in these systems.

We propose the MEC-supported sELf-adaptive eLAsTiCiTy (REACT), a new auto-scaling strategy that addresses the previously described weaknesses of reactive approaches employing a solidarity-based elasticity algorithm. REACT is an alternative strategy to avoid service migration due to resource scarcity. Namely, we propose that telecom operators can re-allocate network and MEC resources from over-provisioned services and redistribute them to under-provisioned services while keeping all services running on the same server. REACT distinguishes itself from reactive elasticity solutions in three ways: (i) optimal auto-scaling of both network-level and compute-level virtual resources at network edges under resource scarcity conditions; (ii) efficient resource allocation of over-provisioned resources from a set of *donor* services to scale-up demanding *recipient* services; and (iii) self-adaptive auto-scaling, which reduces the *elasticity attempt window* during the scarcity of MEC resources.

REACT can be implemented for several use cases. We highlight two use cases and how REACT could work around their problems. 1. **Mobility-induced services:** During users' mobility, latency-sensitive services are forced to be migrated through MEC servers. Thus, MEC servers must avoid blocking of service migration, which can occur due to MEC re-

source scarcity. Besides, it contributes to service QoS degradation in mobility environments since the edge server selection can disregard an optimal edge server due to its workload. MEC servers must meet all service migration requests to ensure the deployment of mobility-induced strategies in real-edge environments. REACT can address this problem through its solidarity approach. REACT could guarantee that latency-sensitive services would obtain resource reservations in MEC servers during these service migrations, classifying these services as priority services, i.e., *recipient services*, and the other services implemented in the MEC infrastructure as *donor services*. 2. **Service billing:** The *pay-as-you-go* model has been implemented in cloud computing and will be incorporated during edge deployments. The more resources are used, the more the tenant has to pay. In this case, over-provisioned resources can increase the price paid to host services and applications both on the cloud and at the edge. REACT can work around this problem by over-provisioned resource optimization. REACT uses over-provisioned resources to support new auto-scaling requests rather than requesting new resource allocations to the cloud or edge provider. It can then prevent tenant's expenses from increasing in edge computing.

The rest of this paper is structured as follows. Section 2 presents the most relevant related works, highlighting their strengths and shortcomings. Section 3 presents the system model. Section 4 details REACT's architecture and operation. Section 5 presents an experiment setup used to evaluate REACT's performance. Section 6 evaluates the experimental results and comments them. Finally, Section 7 summarizes the article's findings.

## 2. Related Work

Several studies [4, 16, 17] have investigated alternative approaches for resource elasticity in cloud computing and they conclude that the scarcity of resources cannot negatively impact services running on large cloud providers, e.g., Amazon EC2, Azure, GCP. Compared to large-scale cloud systems, a MEC server can provide lower communications delay between user and server, but it also comes with less resources than cloud infrastructures. The scarcity of MEC resources may affect service performance because some under-provisioned services might need to be migrated to another MEC server, introducing service-restart delays in some cases.

One of the most popular container orchestration tools used by network operators to support cloud computing is Kubernetes<sup>1</sup>. The massive infrastructure investments by network operators drive the move to Kubernetes, enabling containerization in the cloud and at the edge network to afford 5G MEC services based on lightweight virtualization deployments. The Kubernetes architecture consists of a logical master node, which can deploy a variable set of application containers called *pods* on a group of nodes. In order to allocate system resources to the running services, Kubernetes follows the *auto-scaling* principle, which proposes to reactively increase or decrease the resources allocated to the

<sup>1</sup><http://kubernetes.io>

service according to its current demand. One way in which Kubernetes can adjust the resources allocated to a service is by increasing or decreasing the resources associated to each pod, through a module named Vertical Pod Autoscaler (VPA). The VPA estimates every pod's resource utilization and, if their current workloads go beyond a threshold, it restarts the resource-intensive services granting them a more suitable amount of resources. If resources are not available on the current server, where the service is already deployed, the VPA redeploys the service to another server. One drawback of restarting or migrating the pod is that stateful context information must be copied between two replicas (in case of a make-before-break approach) or at least stored and reloaded (in case the server does not allow the creation of another pod before tearing down the old one). While Kubernetes uses migration in case of scarce resources, REACT tries to reallocate over-provisioned resources to avoid service migration. Hence, Kubernetes' auto-scaling policy reduces the resource allocation efficiency under resource scarcity conditions because it triggers several resource-reallocation rounds.

Due to resource limitations imposed by MEC servers compared to large-scale cloud providers, a few works have investigated resource elasticity in edge networks [19]. For example, Yuan et al. [20] propose a scheme to serve the time-varying demand for resource capacity from mobile services. The proposed solution deploys online Virtual Network Function (VNF) scaling, which realizes an on-demand resource allocation in MEC infrastructures. Wang et al. [21] propose a framework to manage edge nodes and an auto-scaling mechanism for resource provisioning in edge nodes, which is based on three stages, i.e., handshaking, deployment, and termination. Righi et al. [22] present the Elastic-RAN model, which proposes multi-level and adaptable resource elasticity for Cloud Radio Access Networks. Adaptivity refers to the elasticity level in which physical machines and their resources are provisioned as close as possible to the current processing needs. Authors in [23] have proposed an auto-scaling algorithm to minimize costs and deal with unbalanced cluster load caused by resource expansion, i.e., scale-up, and the data reliability caused by resource scale-down. The work in [24] proposes a VM-scaling algorithm to Distributed Enterprise Information Systems, which optimally detects the most appropriate scaling conditions using performance-models of distributed applications based on SLA-specified performance constraints. Naha et al. [25] developed resource allocation and provisioning algorithms by using resource ranking and provisioning of resources in a hybrid and hierarchical fashion to address the problem of satisfying deadline-based dynamic user requirements in fog computing. These works focus on QoS maintenance at MEC infrastructures. However, they always consider available resources to support the required elasticity demand. Kumar et al. [17] claim that SLA violations need to be detected in the resource provisioning process when resource elasticity issues on cloud and edge servers happen. This can occur under resource scarcity conditions, hence, jeopardizing QoS and Quality-of-Experience (QoE).

Li et al. [26] propose a scheduling optimization mecha-

nism for improving consistency maintenance in edge environments. The mechanism is based on a two-level scheduling optimization scheme. If the edge data center does not have enough resources to complete, it will migrate the service to a centralized cloud data center. Castellano et al. [27] proposed DRAGON, a distributed resource assignment and orchestration algorithm that seeks optimal partitioning of shared resources between different applications running over a standard edge infrastructure. The evaluation allowed testing the algorithm behavior after the hosting resources have been saturated, even running a low number of applications. The work in [28] has proposed an auction-based resource allocation and provisioning mechanism, which produces a map of application instances in edge computing, namely Edge-MAP. Edge-MAP considers users' mobility and the limited computing resources available in edge micro-clouds to allocate resources to bidding applications. Edge-MAP can reallocate resources to adapt to the dynamic network conditions. Guo et al. [29] recommend an on-demand resource provisioning mechanism based on load estimation and service expenditure (over-provisioned resources) for edge cloud. The mechanism uses a neural network model to estimate the resource demand. However, before releasing the node resources, the user data on the node need to be migrated to other working nodes to ensure service continuity. Sarrigiannis et al. [30] proposed a VNF lifecycle management through an online scheduling algorithm, where the VNFs are orchestrated, e.g., instantiated, scaled, migrated, and destroyed, based on the actual VNF traffic. Authors also proposed an experimental evaluation based on the implementation of a MEC-enabled 5G platform. The assessment aimed to maximize the number of served users in MEC by taking advantage of the online allocation of edge resources without violating the application SLAs. Akhtar et al. [31] proposed the management of chains of application functions over multi-technology edge networks. This work provides solutions to resource orchestration and management for applications over a virtualized edge computing infrastructure.

Most of the aforementioned works trigger service migration in resource scarcity situations, which can affect QoS and QoE [3]. Migrating a service has several drawbacks, such as increased latency, traffic congestion and network usage costs, due to the data transferred between remote hosts. In the real world, where multiple network operators manage the infrastructure, migrating a service may take longer than expected because mobile network operators must agree to exchange the service across heterogeneous platforms.

The aforementioned works show that only a few studies in the literature have investigated resource elasticity in MEC, and those who do are characterized by a set of common limitations, detailed hereafter. Firstly, resource elasticity models do not consider the resource scarcity of MEC in their design. Secondly, most related works frequently trigger service migration procedures. Finally, most related works do not optimize MEC resources utilization, resulting in a long *elasticity attempt window*. In this paper, we aim to tackle these three limitations arising from previous works by proposing

**Table 1**

Comparison of related works towards optimal MEC-tailored elasticity. Legend: 1=Constrained capacity, 2=Successful auto-scaling, 3=Elasticity attempts, 4=Self-adaption.

Solutions (References)	Requirements			
	1	2	3	4
Kubernetes VPA	✓			
Yuan et al. [20]	✓			
Wang et al. [21]	✓			
Righi et al. [22]	✓			
Chunlin et al. [23]	✓			
Antonescu et al. [24]		✓		
Naha et al. [25]	✓		✓	
Li et al. [26]	✓		✓	
Castellano et al. [27]	✓		✓	✓
Tasiopoulos et al. [28]	✓		✓	
Guo et al. [29]	✓			
Sarrigiannis et al. [30]	✓			
Akhtar et al. [31]	✓			
REACT (present work)	✓	✓	✓	✓

REACT: a self-adaptive elasticity mechanism as a heuristic solution tailored to MEC resource scarcity conditions.

Based on the literature review, we identify that new approaches need to evolve to tackle resource elasticity among MEC systems while meeting the stringent requirements of 5G applications. This imposes a set of challenges when carrying out elasticity strategies in large-scale MEC scenarios since it cannot accommodate a high density of resource elasticity requests. Thus, it becomes even more problematic by directly affecting 5G applications' performance. Although MEC servers have computing power, with the increase of users, its limited computing power is gradually overloaded, which cannot guarantee the QoS of particular applications. The challenge consists of designing an optimal resource elasticity mechanism to support 5G application requirements.

We claim that MEC characteristics, e.g., resource limitation, lead to the adoption of optimal self-scaling solutions, affording QoS and resource-constrained awareness to keep 5G applications always better served by the underlying MEC facilities [32]. The list of requirements we claim for an optimal solution of a MEC-tailored elasticity mechanism includes the following requirements to be met:

1. Provisioning capacity in MEC environments;
2. Capacity to provide auto-scaling whenever the service needs more resources, employing an enhanced *elasticity attempt window* to respond to new loads;
3. Successful auto-scaling under resource scarcity conditions and decreasing the number of unsuccessful elasticity attempts;
4. Deploying a self-adaptive approach to tackle the issues that widely-used reactive auto-scaling solutions raise.

**Table 2**

Notations and symbols.

Symbol	Explanation
$S$	Set of services running on the MEC server.
$s_i$	$i$ -th service $\in S$ .
$w_i$	Workload of the $i$ -th service.
$a_i$	Resource allocation of the $i$ -th service.
$o_i$	Resource over-provisioning of the $i$ -th service.
$\omega$	Server background workload.
$\xi$	MEC server load.
$\beta$	Auto-scaling for a service $s$ in the MEC server.
$\delta$	Set of service monitoring metrics.
$h$	Monitoring metric of service $s_i$ , where $h_i \in \delta$ .
$R$	Recipients list.
$D$	Donors list.
$r$	Recipient service, where $r \in R$ .
$d$	Donor service, where $d \in D$ .
$\mu$	A function that represents the <i>donation</i> from a donor service $d$ to a recipient service $r$ .
$T_c$	Committed service threshold.
$T_d$	Service donating threshold.

Table 1 compares the main characteristics of the related works concerning the aforementioned requirements and shows that none of the considered solutions can support all our claimed requirements towards optimal auto-scaling. Motivated by the limitations of the reactive approaches of related works, we propose the REACT solidarity-based elasticity strategy, as described in the next section.

### 3. System Model

The considered MEC infrastructure consists of a set of interconnected MEC servers, each of them offering different computing and memory resources to a set of running services, each having distinct and specific resource requirements. We assume that each MEC server's workload is modeled as a quadruple representing only four types of available resources: computation, communication, main memory, and permanent memory, whose amounts do not change over time. Since REACT redistributes resources among the services running on a single MEC server, we restrict our scope to a set  $S$  of running service instances on a single MEC server. We assume that the time in the system is divided into equal intervals called time slots, and the system produces a service resource reallocation during each time slot. REACT operates within a single time slot, so we assume that all the symbols introduced hereafter are related to a certain time slot  $k \in \mathbb{N}$ .

We define the *server background load*  $\omega \in [0, 1]^4$  as a quadruple that represents the resource load on the MEC server unrelated to running user services, e.g., OS overhead, scheduling, background and monitoring processes, which cannot be auto-scaled. We define the MEC server load  $\xi$  as the sum of the background load  $\omega$  and the total amount of resources allocated to all services running on MEC server. Equivalently,  $\xi = \omega + \sum_{i=1}^{|S|} a_i$ . It is worth noting that  $\forall k \in \mathbb{N}, 0 \leq \xi \leq 1$ , as the sum of the allocated resources for the



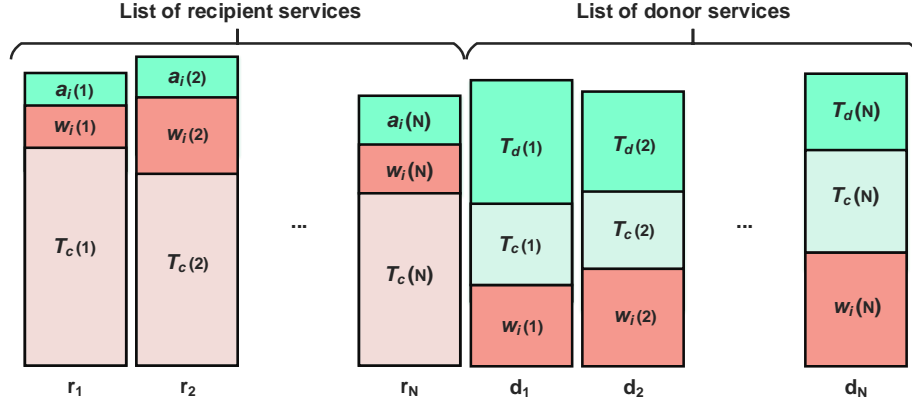


Figure 1: System model

services and the background processing on the MEC server can never exceed its maximum resource capacity.

MEC servers' resource utilization can be classified into three categories: light, medium, and heavy utilization. A MEC server is under light utilization if its  $\xi \leq \tau_l$ , where  $\tau_l \in [0, 1]$ . Similarly, a MEC server is under heavy utilization if its  $\xi \geq \tau_h$ , where  $\tau_h \in [0, 1]$ . If  $\tau_l < \xi < \tau_h$ , then the MEC server is under medium utilization.  $\tau_l$  and  $\tau_h$  represents 30% and 95% of the MEC server's capacity, respectively. The low and high thresholds will determine the when REACT will trigger its solidarity approach. We consider that a MEC server is in a *resource scarcity* condition when its  $\xi > 0.95$ .

Every service  $s_i \in S$  running on the system is characterized by a set of parameters, detailed hereafter. The *workload* of service  $s_i$  is indicated with  $w_i \in [0, 1]^4$ , a quadruple in which each element represents the ratio between the service's current load and the MEC server's capacity for a specific resource type. The *resource allocation* of service  $s_i$  is indicated with  $a_i \in [0, 1]^4$ , a quadruple in which each element represents the ratio between the amount of resources allocated for service  $s_i$  and the MEC server's capacity for a specific resource type. The *resource over-provisioning* of service  $s_i$  is defined as  $o_i = a_i - w_i$ , a quadruple in which each element represents the ratio between the amount of over-provisioned resources for service  $s_i$  and the MEC server's capacity for a specific resource type.

REACT classifies every service as either *donor service* or *recipient service*. A donor service  $d$  is defined as an over-provisioned service that is willing to transfer part of its currently unused resources to other services that need them. A recipient service  $r$  is defined as a service that is currently under-provisioned and close to run out of resources, which is willing to accept resources from other donors.

REACT's solidarity approach considers that a set of recipients  $r$ , under resource scarcity conditions, are eligible for receiving resources from other over-provisioned donors  $d$  that run on the same MEC server. Donors scale-down parts of their over-provisioned resources to scale-up recipients. As long as services have residual resources, REACT remains able to auto-scale recipients and avoid Service-Level

Agreement (SLA) violations. The computation performed by REACT to decide the amount of over-provisioned resources to transfer from a set of donors  $d$  to each recipient  $r$  is called *donation*.

The *committed service threshold*  $T_c(s_i)$  is the minimum amount of resources needed by the service  $s_i$  to honor its SLAs. We define the *service donating threshold* as  $T_d(s_i) = a_i - T_c(s_i)$  as the maximum amount of resources that service  $s_i$  can donate.  $T_d(d)$  quantifies the part of the donor's over-provisioned resources  $o_d$ , aiming to scale-down donors and scale-up recipients. The expression for  $T_d$  is designed so that a donor  $d$  cannot donate more resources than what its SLA allows it, when  $w_d \leq T_c(d)$ . Figure 1 shows the thresholds  $a_i$ ,  $w_i$ ,  $T_c$ , and  $T_d$  for each service in the system, where each variable is used to represent recipients  $r$  or donors  $d$  in the solidarity-based model.

Let us define  $q$  as a decision binary variable, where  $q \in \{0, 1\}$ , assumes value 1 to perform scale-up and 0 to perform scale-down. The resource type that will be scaled up/down is denoted by  $\gamma \in \{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$ . The share of resources that will be scaled up/down is denoted as  $z \in [0, 1]$ . The auto-scaling function for a service  $s_i$  represents the amount of resources that the service will either receive or donate, and is denoted as  $\beta(s_i, \gamma_i, q, z) = \gamma \cdot (1 + (2q - 1)z)$ .

The total amount of resources exchanged in a *donation* from a set of donors  $D' \subset D$  to a specific recipient  $r \in R$  for a specific resource type  $\gamma$  can be defined as  $\mu(r, D', \gamma) = \sum_{s \in D'} \beta(s, \gamma, 0, T_d(s))$ . If the donation process involves a set of recipients  $R' \subset R$  and a set of donors  $D' \subset D$ , then the amount of exchanged resources can be computed as:

$$\sum_{s \in R'} \left( \mu(s, D', \gamma) + \sum_{s \in D'} \beta(s, \gamma, 0, T_d(s)) \right) \quad (1)$$

The donation for a specific recipient  $r$  occurs until the sum of scale-down resources from a set of donors  $d \geq T_m(r) \cdot 1.3$ . The value of  $z$  for the  $i$ -th  $r$  in each donation procedure is set to 30%. Each donation adds 30% more resources than the current  $w_r$  in time slot  $k$ . We scale-up each  $T_m(r)$  by 30% to avoid new donation requests in a short time period. According to

our analysis and the thresholds practices adopted in [33], we chose 30% as the threshold. It mitigates the over-provisioning and improves the time window in which the service will need another auto-scaling procedure. On the other hand, the value of  $z$  for the  $i$ -th  $d$  is set to its  $T_d$ . Hence, for any donation procedure, the property  $\sum_{s \in D'} T_d(s) \geq T_m(r) \cdot 1.3$  holds. It is noted that each  $T_m(r)$  is updated via  $\mu$ . Thus, Equation 1 minimizes the over-provisioned resources in MEC servers and maximize resource utilization. We want to maximize resource utilization as long as we can satisfy the elasticity demands and do not violate SLAs.

Let us define  $h_i = (w_i, a_i, o_i)$  as the *monitoring metrics* of the  $i$ -th service, i.e., the current values for its workload  $w_i$ , allocated resources  $a_i$ , and over-provisioned resources  $o_i$ . Each service monitoring metric  $h_i$  uses  $\gamma$  to denote the types of resources for a service  $s_i$ , e.g., CPU, RAM, storage, and bandwidth. We can then define  $\delta$  as the set of service workloads deployed in a generic MEC server, where  $h_i \in \delta$ . A MEC server uses  $\delta$  to obtain the full service status information, then  $\delta = \sum_{i=1}^n h_i$ , assuming that the server must check each service serially. In the considered scenario, we assume that the value of  $\delta$  is updated periodically. The frequency with which  $\delta$  is updated significantly influences REACT's behavior, as service monitoring is a crucial measure to determine whether the solidarity-based approach should be triggered. Table 2 lists the key parameters of the system model.

## 4. REACT

This section describes the principles of REACT, its architecture and how it operates, including the solidarity-based elasticity algorithm and its complexity analysis.

### 4.1. REACT Architecture

The efficiency behind an elasticity mechanism depends on the auto-scaling function. As edge services' requirements change over time, MEC servers will experience workload fluctuations. These workload fluctuations may result in either service over- or under-provisioning. When the load decreases, the most widely adopted reactive mechanisms will take some time to provide scale-down actions. On the other hand, auto-scaling mechanisms will scale-up and cause over-provisioning when the load increases. If resources are scarce, it will cause under-provisioning. The over-provisioning strategy reserves more resources than those needed by the service at a specific moment in time, aiming to avoid disruptions, if the service requires an unexpectedly high amount of resources to support its operations in the future.

Over-provisioning demands careful deployment to prevent the inefficient resource allocation. However, in situations where over-provisioned resources are low, reactive auto-scaling solutions tend to trigger several elasticity rounds until matching resource patterns to meet the new service workload, which increases the *elasticity attempt window*. Even though this strategy will ensure that SLAs are not violated, it might reserve resources for services, which in turn may never use them. This would lead to inefficient MEC resource usage and unnecessary costs for the user to benefit from those MEC

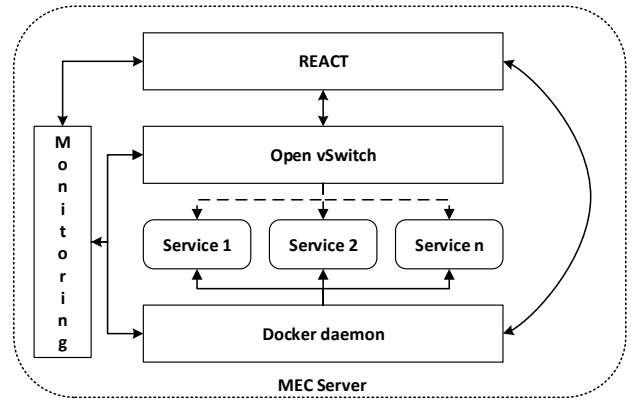


Figure 2: REACT Architecture

resources that do not positively impact the application's QoS. In under-provisioning, the allocated resources for a given service are less than the current load demand, which can cause SLA violations and service resizing penalties.

REACT provides an auto-scaling algorithm to efficiently reallocate resources among different services running on MEC servers under scarce resources. REACT solves the typical problems of reactive schemes, e.g., several auto-scaling rounds during resource scarcity situations, by re-orchestrating both networking and computational MEC resources. The main novelty of REACT, compared to other reactive resource elasticity mechanisms, is its solidarity-based resource reallocation, which defines how some resources are seized from a set of donors and transferred to a set of recipients when the system enters a resource-depletion state.

REACT's solidarity-based elasticity takes advantage of services' resource over-provisioning to offer enhance auto-scaling capability towards MEC efficient resource usage. In contrast, reactive solutions suffer from over-provisioning by needing successive attempts until matching the required resource amounts to the new service load when resources become scarce. It is worth noting that REACT can apply its solidarity scheme only if the MEC server is running over-provisioned services while the available resources in the system become scarce. REACT aims at mitigating the service degradation due to the unavailability of resources in MEC servers and at improving system efficiency by reducing over-provisioned resources. This resource reduction can also decrease the economical costs sustained by the user, since cloud systems provide resources based on a *pay-as-you-go* pricing.

REACT is implemented as part of the auto-scaling component's logic without MEC architectural changes, e.g., the need for adding new components, interfaces, and protocols. Its solidarity-based model can be deployed in any platforms that support auto-scaling mechanisms, making REACT an agnostic solution to MEC servers. Figure 2 presents the REACT architecture, where REACT uses its solidarity algorithm to provide resource reallocation and a monitoring system to check both MEC and service workloads. Furthermore, REACT uses both *Docker* and *Open vSwitch* APIs to reallocate computing and network resources between services.

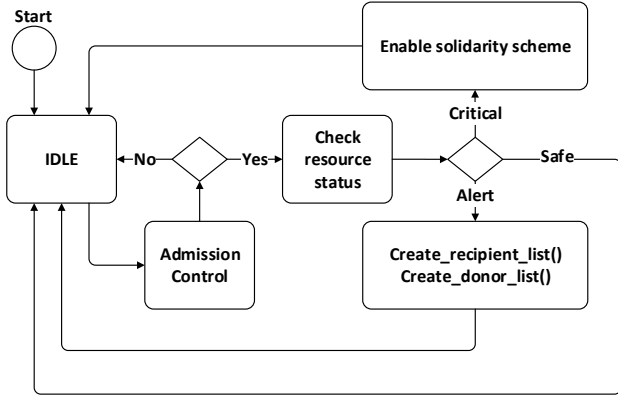


Figure 3: Conditions to enable the REACT solidarity approach.

REACT classifies a server's load into three conditions: *safe*, *alert*, and *critical*. Safe and critical conditions are mapped to  $\tau_l$  and  $\tau_h$ , respectively. The alert condition is enabled when the MEC server load  $\xi$  is between 80% and 95% of the MEC server's capacity. If the system is in safe conditions, REACT does not operate because services can be deployed immediately. When the system is in alert or critical condition, REACT takes preventive measures to reallocate resources and avoid that the system enters or remains in a critical condition. Figure 3 illustrates the conditions for enabling the solidarity approach in a state diagram.

REACT groups services into a *donor list*  $D$  and a *recipient list*  $R$ , respectively. REACT adds a service  $s_i$  to the donor list if its workload  $w_i \leq T_c(s_i)$ . The donor list and the recipient list are sorted from the smallest to the largest available residual resources and resource demands. REACT constantly maintains the recipient list and the donor list if the server reaches an alert or critical condition. Each donation involves a single recipient and one or more donors: after REACT calculates how many resources a single recipient needs, it will scale-down one or more donors and subsequently scale-up the recipient to fulfill its resource needs. REACT will start a donation process until either the  $R$  or the  $D$  is empty.

## 4.2. REACT Operation

On a generic MEC server, the REACT algorithm runs on a set of services  $S$ . First, REACT gathers the infrastructure and service monitoring data, e.g., CPU, RAM, storage, incoming and outgoing bandwidth, to create and maintain the recipient list  $R$  and the donor list  $S$ . We implement  $R$  and  $D$  as self-balancing binary search trees, i.e., *AVL tree*, aiming to optimize the solidarity auto-scaling algorithm. To access  $n$  service monitoring metrics  $h$  REACT uses  $\delta$ . Then, both lists are inspected to meet the highest-priority services that experience resource bottlenecks. After this, REACT calculates the details of the service donations and update the new  $a_r$  and  $a_d$ , respectively, in  $R$  and  $D$ . The next step is to update the service thresholds in both  $R$  and  $D$  lists deployed at the local MEC server. It can be implemented through virtualization platform used to host the service components, e.g., Xen, KVM, Docker Swarm, Kubernetes, OpenVZ.

### Algorithm 1: Recipient and donor service selection

**Input:** service\_list  
**Output:**  $R, D$

```

1 Function ServiceSelection(service_list):
2   Function InsertAVL(root, node)
3   for  $s$  in service_list do
4     if  $w_s \geq T_c(s)$  then InsertAVL( $R, s$ )
5     else InsertAVL( $D, s$ )
  
```

### Algorithm 2: Solidarity-based auto-scaling

**Input:** service\_list  
**Output:**  $R, D$

```

1 Function SolidarityAutoScaling():
2   Function MECMonitoring( $\delta, \xi$ ): Start  $\delta$  and  $\xi$ 
3   while  $\xi \geq \tau_h$  do
4      $R, D \leftarrow \text{ServiceSelection}(\text{service\_list})$ 
5     Function InOrder( $R$ ):
6       if  $R$  is NULL then return
7       InOrder( $R \rightarrow \text{left}$ )
8     Function ReverseOrder( $D$ ):
9       if  $D$  is NULL then return
10      ReverseOrder( $D \rightarrow \text{right}$ )
11      required_donation  $\leftarrow 1.3 \cdot a_R$ 
12      if  $w_D \leq T_c(D)$  then
13         $T_d \leftarrow a_D - T_c(D)$ 
14        donation( $R, D, T_d$ )
15      else
16         $T_d \leftarrow a_D - w_D$ 
17        donation( $R, D, T_d$ )
18      Function Remove( $D$ )
19      if donations  $\geq$  required_donation then return
20      ReverseOrder( $D \rightarrow \text{left}$ )
21      InOrder( $D \rightarrow \text{right}$ )
  
```

Algorithm 1 identifies services that are facing resource bottlenecks, i.e.,  $R$ . Also, it defines the function *InsertAVL*(*root*, *node*) to insert nodes in an AVL tree (line 2). Based on this algorithm,  $R$  and  $D$  lists are created and maintained by Algorithm 1. A service is classified as  $R$  if its workload  $w_s \geq T_c(s)$  (line 4). Algorithm 1 identifies services that can be part of the donation process provided by REACT. A potential  $D$  can be identified by inspecting service workload  $w_s < T_c(s)$  (line 5). In the end,  $R$  and  $D$  are already sorted according to the resource needs and the number of residual resources available, respectively. Algorithm 1 is triggered before a *critical* resource condition has been reached and then after the solidarity scheme is enabled.

Algorithm 2 is triggered as an infinite loop. Each iteration of Algorithm 2 requires getting the service and MEC monitoring metrics (line 2). *Critical* conditions can be identified by checking the MEC load (line 3). Every time a critical resource

condition has been reached, the REACT approach is enabled. REACT builds and maintains both R and D through Algorithm 1 (line 4). In lines 5 and 8, the REACT algorithm defines functions  $\text{InOrder}(\text{root})$  and  $\text{ReverseOrder}(\text{root})$  to recursively iterate over R and D, respectively. On one hand,  $\text{InOrder}(\text{root})$  traverses the left *subtree*, visits the *root*, and traverses the right *subtree*. On the other hand,  $\text{ReverseOrder}(\text{root})$  traverses the right *subtree*, visits the *root*, and traverses the left *subtree*. Line 11 gets the required donation from a set of R. In lines 12 and 15, the algorithm gets the value of  $T_d(d)$ . In lines 13 and 16, Equation (1) is used to re-orchestrate R and D. After the donation of  $T_d(d)$ , the donor  $d$  is removed from D using function  $\text{Remove}(\text{D})$  in line 17. The recursive function in line 8 is either triggered until the required donation is reached or when D is empty (line 18).

To prove the feasibility of implementing the REACT solidarity approach in real-time MEC servers, we provide a detailed algorithm complexity analysis. To give an accurate analysis, let us assume that: (i)  $n$  services are running on MEC server; (ii)  $n$  services are classified as donor (D) and recipient (R) services; and (iii) on average, the REACT solidarity scheme consists of 30% of R and 70% of D.

Although  $n$  services are iterated/searched in line 3 with complexity  $\mathcal{O}(n)$ , lines 4 and 5 use AVL tree insertion function  $\text{InsertAVL}(\text{root}, \text{node})$ , which has time complexity  $\mathcal{O}(\log n)$ . Since lines 4 and 5 of Algorithm 1 are not nested, we can derive that Algorithm 1 has time complexity  $\mathcal{O}(n \log n)$ .

Algorithm 2 gets MEC and service monitoring metrics in line 2 through function  $\text{MECMonitoring}(\delta, \xi)$ , which has time complexity  $\mathcal{O}(n)$ . Algorithm 2 uses a *while* loop in line 3 to enable the REACT solidarity model, where in each iteration the MEC workload  $\xi$  is updated. Line 4 has time complexity  $\mathcal{O}(n \log n)$  as it uses Algorithm 1. Within function  $\text{InOrder}(\text{R})$ , in line 8, the function  $\text{ReverseOrder}(\text{D})$  has time complexity  $\mathcal{O}(d)$  as it recursively iterates over D. Within function  $\text{ReverseOrder}(\text{D})$ , in line 17 the function  $\text{Remove}(\text{D})$  performs  $\mathcal{O}(1)$  as it already uses  $\text{ReverseOrder}(\text{D})$  to find the node. Then,  $\text{Remove}(\text{D})$  removes the donor  $d$  from D and performs the AVL rotations when needed. As R and D have a linear relationship with  $n$  and based on  $\text{ReverseOrder}(\text{D})$  and  $\text{Remove}(\text{D})$  algorithm analysis, which are nested and within function  $\text{InOrder}(\text{R})$ , in line 5 the function  $\text{InOrder}(\text{R})$  has time complexity  $\mathcal{O}(n^2)$  as it takes  $\mathcal{O}(r)$  to recursively iterates over R, resulting in the product  $\mathcal{O}(r) \cdot \mathcal{O}(d) \cdot \mathcal{O}(1)$  for searching in R, D, and removing from D, respectively.

For both  $\text{InOrder}(\text{R})$  and  $\text{ReverseOrder}(\text{D})$ , the comparisons during the search in each iteration, including unsuccessful search, are limited by the height of the AVL tree, which is  $\mathcal{O}(\log n)$ . As  $\text{InOrder}(\text{R})$  and  $\text{ReverseOrder}(\text{D})$  have to search all nodes, then both perform  $\mathcal{O}(n)$ .  $\text{InsertAVL}(\text{root}, \text{node})$  requires  $\mathcal{O}(\log n)$  to lookup a service, plus a maximum of  $\mathcal{O}(\log n)$  retracing levels on the way back to the *root*, which takes  $\mathcal{O}(\log n)$ .  $\text{Remove}(\text{D})$  follows the same pattern of function  $\text{InsertAVL}(\text{root}, \text{node})$ , which also has time complexity  $\mathcal{O}(\log n)$  [34]. However, as it is used within  $\text{ReverseOrder}(\text{D})$ , it already knows where the node is, just requiring  $\mathcal{O}(1)$  to remove the node and perform the AVL rotations.

As  $\text{MECMonitoring}()$ ,  $\text{ServiceSelection}()$ , and  $\text{InOrder}()$  are not nested, the function  $\text{SolidarityAutoScaling}()$  has time complexity  $\mathcal{O}(n^2)$ . We conclude that the REACT algorithm performs  $\mathcal{O}(n^2)$  resource reallocation operations.

## 5. Experiment Setup

To assess their impact in handling elasticity events, both Kubernetes and REACT adopt the same elasticity approach to scale-up/down resources of MEC services. When a service reaches the resource utilization threshold of 70%, both mechanisms scale-up by 30% of the current service resource allocation. Otherwise, when the current service resource usage is  $\leq 30\%$ , they perform a scale-down of 20% of the allocated resources. These thresholds are commonly used in other approaches and considered as good practices for cloud computing [33]. If the vertical elasticity cannot be achieved successfully, Kubernetes will ignore the elasticity event. In contrast, REACT triggers the solidarity elasticity mode.

To denote a MEC-like testbed, we design the testbed configuration as described in Figure 4. The auto-scaling schemes have been implemented in an Openstack-based cloud platform, consisting of three Dell power edge servers, two external Dell PowerVault md3800i that provide disk space of 20.6 TB in RAID 5, and a network backbone with 48x10 GbE-T ports and 80 Gbit/s backbone connection.

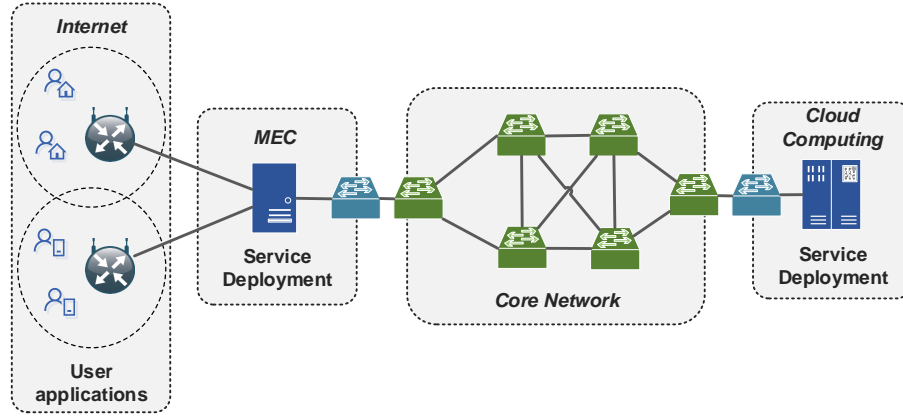
We represent edge servers as virtual machines deployed on our MEC infrastructure. Each edge server uses Ubuntu server 18.04.4 as an operating system, with 8 vCPUs and 16GB RAM. Moreover, Docker (version 19.03.8) and Open Virtual Switch (OVS) (19.03.8) are used for the software stack. Furthermore, a monitoring stack composed by Prometheus<sup>2</sup> and cAdvisor<sup>3</sup> technologies are used to book VM and container-level resource usage and performance. Prometheus provides the node exporter to get the VM monitoring metrics, and cAdvisor gets the container monitoring metrics. Edge services are deployed to run in Docker containers, whereas the OVS provides a virtualized network infrastructure interconnecting the participating MEC and cloud servers.

The auto-scaling solutions used in the experiments, i.e., REACT and Kubernetes VPA, leverage the Docker and OVS APIs to scale up/down computing, i.e., vCPUs, RAM, and network, i.e., bandwidth, resources, respectively. This auto-scaling mechanism provides functions to automatically sets the container resource. These functions are *request* and *limits*. It uses the *requests* and *limits* functions to control CPU and memory resources. VPA seeks to reduce the overhead of setting resource requests and limits for containers and improve cluster resource usage. The main features of VPA are: (i) reduce the request value for containers whose resource usage is consistently lower than the requested amount; (ii) increase request values for containers that consistently use a high percentage of resources requested; and (iii) automatically set containers' resource limit values based on request ratios specified as part of the container template/blueprint.

<sup>2</sup><https://prometheus.io/docs/introduction/overview/>

<sup>3</sup><https://github.com/google/cadvisor>





**Figure 4:** The testbed deployment for REACT and Kubernetes experiments.

The Kubernetes VPA algorithm has only CPU and RAM built-in manageable resources by design. We focus on the *limits* function to ensure that a container's resource threshold never exceeds. Also, we provide an elasticity policy to trigger network elasticity events when the resource utilization reaches 80% of reserved resources. We apply the Poisson distribution results in the OVS, where we allocate different bandwidth demands for each service. This feature is incorporated into Docker containers through OVS, where we set virtual tunnels for each container's virtual interface. Furthermore, we set QoS egress and ingress traffic shaping policies to ensure bandwidth limitations for each service deployed within Docker containers.

A set of 100 services is deployed in the edge server, including edge analytic services, Internet of Things (IoT) services, and video services to provide dynamic behavior in a real environment. The edge server has 16 GB RAM, 8 vCPUs, and a 5 Gbit/s link. The client arrival times are modeled by a Poisson process for both REACT and Kubernetes. The elasticity time windows and service parameters such as workload, resource allocation, and over-provisioning are also modeled by a Poisson distribution.

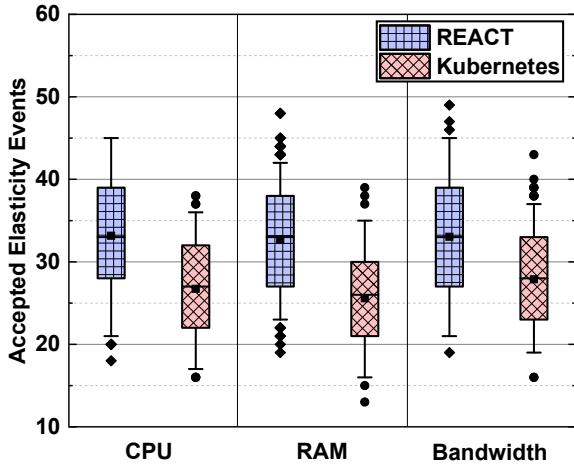
We define the *elasticity time window* as the time required to trigger service elasticity events, i.e., an elasticity event is triggered at time slot  $k$  and in time slot  $k + 1$  another elasticity event is triggered. Then, the workload variations are triggered according to the *elasticity time window*. In total, 1000 elasticity events are generated based on each service's Poisson distribution. Lastly, in our evaluation, we consider that 1 vCPU represents 1024 CPU cycles per second. We use the docker flag `--cpu-shares` to control the CPU allocation priority.

To validate the approach presented in this paper, we implemented a REACT prototype, available at [35] as open-source. The workload generated based on Poisson distribution allowed us to test both REACT and Kubernetes algorithm performance after the MEC resources became scarce. All tests have been repeated along with 1000 elasticity events. Both REACT and Kubernetes are evaluated using the following Key Performance Indicators (KPI):

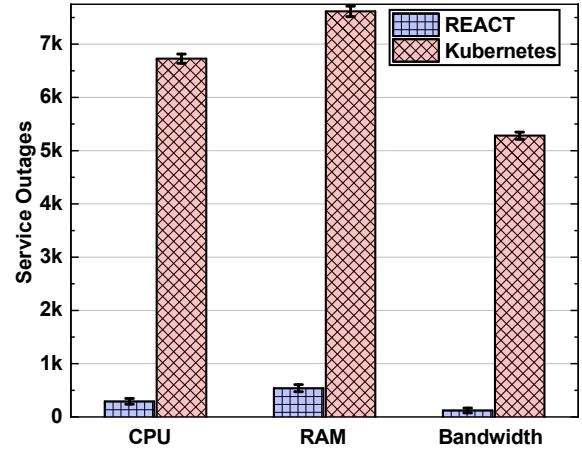
1. **Elasticity events accomplishment** measures both mechanisms' performance to accept elasticity events under resource scarcity conditions. Thus, auto-scaling requests can be denied if no resources are available.
2. **Cumulative Distribution Function (CDF)** shows the cumulative acceptance ratio's behavior along with KPI 1 in the experiment. It shows how REACT can handle more auto-scaling requests than Kubernetes by using its service donation approach.
3. **Service outages** measure the negative impact on services when resources become scarce. Moreover, this KPI shows how services could be either terminated or migration could be enabled due to scarcity of resources.
4. **Elasticity attempts** are related to the algorithmic capacity to calculate new elasticity enforcement during resource scarcity conditions. If no resources are available, a single auto-scaling request will count as one elasticity attempt. The mechanisms will then attempt to respond to the auto-scaling request until resources become available, while elasticity attempts are counted.
5. **Residual resource behavior** (over-provisioned) shows how over-provisioned resources are allocated during the experiments. Based on this, it is possible to understand how resource allocation could be enhanced whenever MEC resources become scarce. Besides, it identifies how service billings can be minimized while providing better MEC resource usage.
6. The **time response** measures both mechanisms' performance to calculate and perform auto-scaling events.

## 6. Performance Evaluation

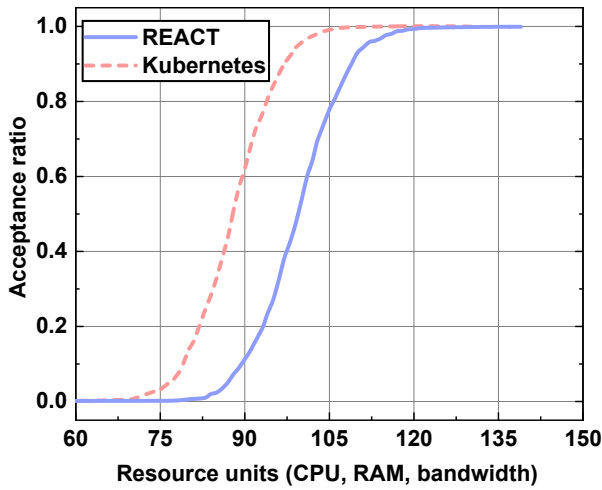
REACT and Kubernetes acceptance elasticity events rates (KPI 1) have been evaluated by measuring the number of events accepted after the hosting resources are saturated. Accepted events are related to both mechanisms' capacity to accomplish elasticity events, e.g., given an elasticity request,



**Figure 5:** Impact of REACT and Kubernetes mechanisms to accomplish elasticity events throughout the testbed.



**Figure 7:** Influence of REACT and Kubernetes elasticity mechanisms in the testbed concerning service outages.



**Figure 6:** Acceptance ratio of elasticity events.

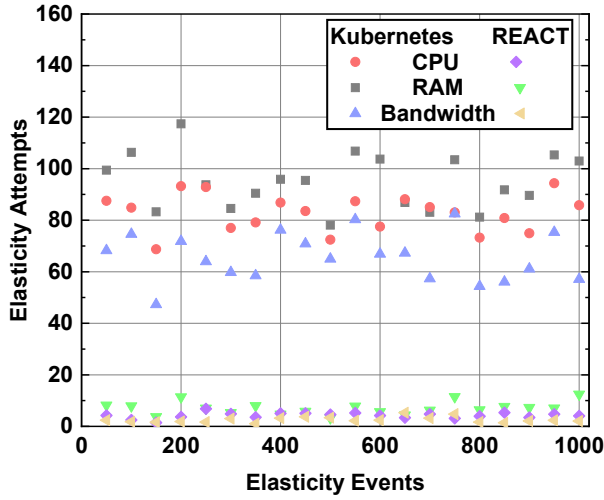
the mechanism can provide the auto-scaling provisioning action. In particular, Figure 5 shows the total accepted elasticity events by each resource type, i.e., CPU, RAM, and bandwidth. Kubernetes achieved an acceptance rate of 80'177 events. Based on this, 33.34%, i.e., 26'733, of the events were dedicated for CPU resources, 31.89%, i.e., 25'568 events, for RAM resources, and 34.77%, i.e., 27'876, for network resources. On the other hand, REACT achieved an acceptance rate of 98'848 elasticity events, where 33.56%, i.e., 33'168 events, for CPU resources, 33.02%, i.e., 32'644 events, for RAM resources, and 33.42%, i.e., 33'036 events, for network resources. REACT has accepted 18'671 more events than Kubernetes, which means a performance gain of 18.88% compared to Kubernetes. It is worth mentioning that the present evidence relies on REACT's capacity to accommodate more elasticity events through its solidarity approach.

We show the acceptance ratio of elasticity events in Figure 6 through a CDF (KPI 2). Also, Figure 6 combines all acceptance probability values, i.e., CPU, RAM, and bandwidth,

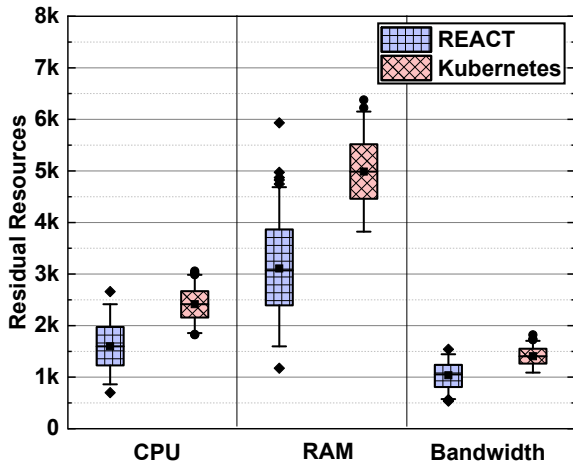
and shows the cumulative probability of the elasticity events accepted by REACT and Kubernetes. REACT has a higher acceptance ratio due to its knowledge of over-provisioned resources. This feature avoids rejection events and increases the acceptance events ratio.

In containerization-based Docker, CPU is a compressible resource; that is, containers can be throttled by the Linux kernel CPU scheduler if the requested amount is exceeded or the node is overloaded. Once a container reaches the limit, it will continue running. However, the operating system will throttle it and keep restricting it from using the CPU. On the other hand, it is important not to allow a running container to consume too much of the host machine's memory. By definition, RAM is a non-compressible resource. Once a container reaches the memory limit, it will be terminated because of the Out of Memory (OOM) problem, which means that the container's service will be killed. The same behavior occurs in REACT since Docker provides container virtualization for services. Kubernetes was designed to maintain the availability of the entire system. When the system goes into the over-committed state, the Kubernetes may decide to kill a set of pods to restore system stability. Generally, if a pod uses more resources than requested, that pod becomes a candidate for termination. On the other hand, REACT will try to use the residual service resources through its solidarity approach to minimize service outages and reduce service migration.

Figure 7 compares solutions in terms of *service outages* (KPI 3) during the experiments. A total of 19'626 service outage events were accomplished by Kubernetes' VPA mechanism, where 34.28%, i.e., 6'728 events, for CPU, 38.81%, i.e., 7'616 events, RAM, and 26.91%, i.e., 5'282 events, for bandwidth. Based on 1'000 elasticity events, in average, 7.616 services were affected by the OOM problem, which means that at least 8 services would have needed to be migrated to another server, totaling 8% of all services deployed. Furthermore, in average, 6.73% of CPU and 5.28% of RAM service resources were affected by the lack of resources. On the other hand, REACT accomplished 955 service outage



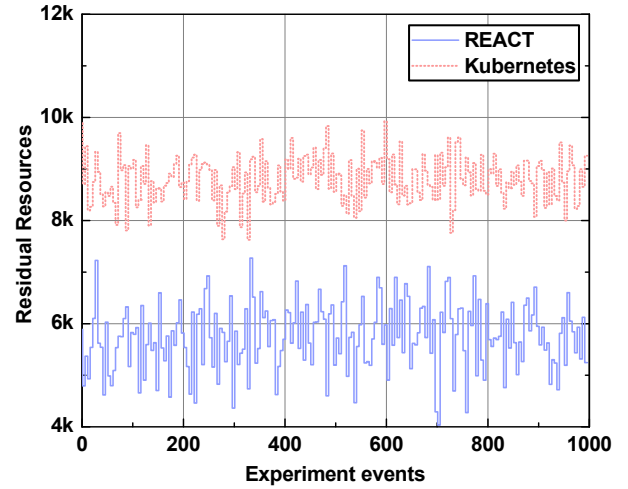
**Figure 8:** Elasticity attempts accomplished in the testbed as a consequence of the REACT and Kubernetes mechanisms.



**Figure 9:** Effect in the residual resources led by REACT and Kubernetes elasticity mechanism on the testbed.

events, equivalent to 4.85% of the total service outage events accomplished by Kubernetes. This means a reduction of approximately 95.15%, i.e., 18'671, of service outage events. For CPU, RAM, and bandwidth resources, REACT detected 293, 540, and 122 service outage events. With REACT, in average, 0.54% of services were affected by the OOM problem. At least 1 service would need to be migrated to another server, totaling 1% of all services. This fact indicates a reduction of 87.5% fewer services affected by the OOM problem than the Kubernetes. These findings support the notion that REACT is less influenced by the OOM problem and, consequently, by the enforced service migration. This implies that REACT is associated with smooth service interruption and prevents more services from becoming terminated or migrated.

Figure 8 shows the performance of both REACT and Kubernetes when the edge server achieves resource saturation, employing the averaging elasticity attempts analysis (KPI 4). When this state is reached, the schemes cannot

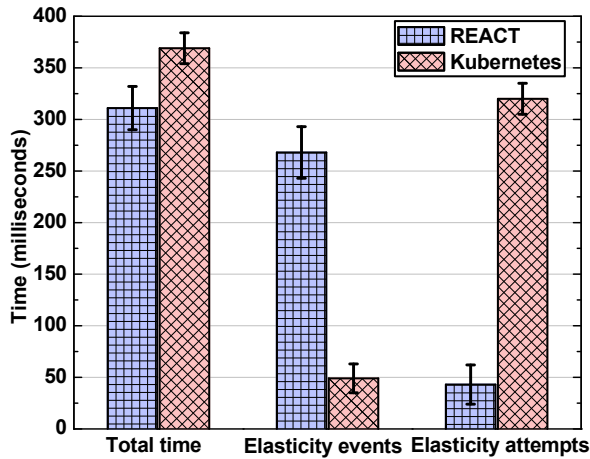


**Figure 10:** Cumulative residual resources behavior led by REACT and Kubernetes elasticity mechanism in the testbed.

serve all service elasticity requests. Then they try to provide elasticity actions based on available resources in the edge server. REACT makes use of the over-provisioned resources. During the resource scarcity situation, Kubernetes achieved 243'456 elasticity attempts, and 34.01%, i.e., 82'811 attempts, of these events were dedicated for CPU resources, 39%, i.e., 94'949 attempts, for RAM resources, and 26.9%, i.e., 65'696 attempts, for bandwidth resources. However, REACT achieved 11'280 elasticity attempts, reducing 95.36%, i.e., 232'176 attempts, compared to Kubernetes elasticity attempts. REACT's mechanism distinguishes itself from the Kubernetes by needing fewer resource re-orchestration rounds to adapt to new loads of services during the scarcity of resources. It chooses a better resource configuration based on all services' workload and can accomplish more elasticity events than Kubernetes.

We also examined the residual resources (KPI 5) for both REACT and Kubernetes. Figure 9 shows the behavior of the residual resources of the mechanisms during the experiment events. Figure 9 sketches the cumulative residual resources units. Kubernetes achieved an average of 2.41 vCPUs cores, residual CPU cycles, 4'985 MB of residual RAM, and 1'404 Mbps of residual bandwidth units. On the other hand, REACT achieved an average of 1.60 residual CPU cycles, 3'070 MB of residual RAM units, and 1'025 Mbps of residual bandwidth units. In this way, REACT performed an average gain of 33.88% of CPU residual resources, 38.41% of RAM residual resources, and 73% of residual bandwidth resources compared to Kubernetes mechanism.

REACT's solidarity algorithm provides scale-down actions on residual resources of the donor list. Figure 10 outlines the residual resource behavior on the elasticity events in the two experiments. Therefore, REACT calculates the ratio between the currently used resources and the total resources reserved for each donor chosen. Then, REACT calculates the final amount of resources to shrink from the residual resources of the selected donor. REACT allows more efficient



**Figure 11:** Processing time that REACT and Kubernetes take in the testbed to accomplish elasticity events.

use of over-provisioning resources by using them more efficiently via the solidarity-based mechanism. REACT takes advantage of over-provisioning, it does not prevent it. The more over-provisioning the MEC services have, the more REACT can make the solidarity approach feasible.

Finally, to evaluate our REACT processing time, we compared the time needed to provide the elasticity actions and the time to provide the elasticity attempts when a resource scarcity situation is reached since both REACT and Kubernetes need to perform restricted actions to meet the current elasticity demand.

Figure 11 shows the average time to process an elasticity request. Figure 11 also outlines the average processing time to accomplish elasticity events and elasticity attempts. Since Kubernetes performs fewer elasticity actions than REACT, its average processing is 49 ms, while REACT achieved 268 ms due to the solidarity actions. Regarding the *elasticity attempt window*, Kubernetes has an average of 320 ms compared to 19 ms of the REACT algorithm, considering that both mechanisms will try to perform elasticity events when resources become scarce. Lastly, the total average processing time, including elasticity events and *elasticity attempt window*, for Kubernetes is 369 ms, while REACT achieved 311 ms. REACT obtained gains in terms of processing time of 15.5% compared to Kubernetes. Indeed, the processing time is short, considering the order of magnitude of milliseconds. Hence, we demonstrate that REACT provides a response time as low as the Kubernetes algorithm.

To summarize the evaluation, REACT compared to Kubernetes has the following improvements: (i) REACT is more agile than Kubernetes, having the ability to accommodate more elasticity events; (ii) REACT provides more resource re-allocation procedures whenever the resources become scarce; (iii) REACT degrades fewer services, allowing services to remain active longer or prevent the service migration; and, (iv) REACT takes advantage of service over-provisioning, enhancing the residual resources of services.

## 7. Conclusion

This paper proposes REACT, a self-adaptive elasticity solution that handles resource scarcity in MEC environments. REACT uses a solidarity approach to provide resource reallocation of residual resources to prevent undesirable service degradation due to the scarcity of MEC resources. REACT can minimize the harmful effects of service migration while keeping more services running over the same MEC server. We provided a detailed description of REACT, including the solidarity approach, the system model, and the REACT algorithm. Our evaluation assesses both REACT and Kubernetes' performance on a real testbed. Testbed results demonstrate better performance of REACT over Kubernetes in terms of accomplishing up to 18.88% more elasticity events, reducing service outages by up to 95.1%, reducing elasticity attempts by up to 95.36%, and reducing over-provisioned resources by up to 33.88%, 38.41%, and 73% for CPU cycles, RAM and bandwidth resources, respectively. Finally, REACT reduced response time by up to 15.5%

## References

- [1] 5G PPP Architecture Working Group. View on 5G Architecture - Version 3.0. Technical report, 5G PPP, 02 2020.
- [2] Imtiaz Parvez, Ali Rahmati, Ismail Guvenc, Arif I Sarwat, and Huaiyu Dai. A survey on low latency towards 5g: Ran, core network and caching solutions. *IEEE Communications Surveys & Tutorials*, 20(4): 3098–3130, 2018.
- [3] Hadeel Abdah, João Paulo Barraca, and Rui L Aguiar. Qos-aware service continuity in the virtualized edge. *IEEE Access*, 7:51570–51588, 2019.
- [4] Tao Chen, Rami Bahsoon, and Xin Yao. A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Computing Surveys (CSUR)*, 51(3):1–40, 2018.
- [5] 5G-PPP Software Network Working Group. From webscale to telco, the cloud native journey. Technical report, 5G PPP, 07 2018.
- [6] Sami Kekki, Walter Featherstone, Yonggang Fang, Pekka Kuure, Alice Li, Anurag Ranjan, Debashish Purkayastha, Feng Jiangping, Danny Frydman, Gianluca Verin, et al. MEC in 5G networks - ETSI White Paper No. 28. Technical report, ETSI, June 2018.
- [7] 5G-PPP Software Network Working Group. Cloud native and 5g verticals services. Technical report, 5G PPP, 02 2020.
- [8] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. *IEEE Communications Surveys & Tutorials*, 19(3):1657–1681, 2017.
- [9] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98: 289–330, 2019.
- [10] Quoc-Viet Pham, Fang Fang, Vu Nguyen Ha, Md Jalil Piran, Mai Le, Long Bao Le, Won-Joo Hwang, and Zhiguo Ding. A survey of multi-access edge computing in 5g and beyond: Fundamentals, technology integration, and state-of-the-art. *IEEE Access*, 8:116974–117017, 2020.
- [11] Cheol-Ho Hong and Blessen Varghese. Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms. *ACM Computing Surveys (CSUR)*, 52(5):1–37, 2019.
- [12] Fei Zhang, Guangming Liu, Xiaoming Fu, and Ramin Yahyapour. A survey on virtual machine migration: Challenges, techniques, and open issues. *IEEE Communications Surveys & Tutorials*, 20(2):1206–1243, 2018.
- [13] Zeineb Rejiba, Xavier Masip-Bruin, and Eva Marín-Tordera. A survey



on mobility-induced service migration in the fog, edge, and related computing paradigms. *ACM Computing Surveys (CSUR)*, 52(5):1–33, 2019.

- [14] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials*, 19(4):2322–2358, 2017.
- [15] Francesco Spinelli and Vincenzo Mancuso. Towards enabled industrial verticals in 5g: a survey on mec-based approaches to provisioning and flexibility. *IEEE Communications Surveys & Tutorials*, 2020.
- [16] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle. Elasticity in cloud computing: State of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2):430–447, 2018.
- [17] Mohit Kumar, SC Sharma, Anubhav Goel, and SP Singh. A comprehensive survey for scheduling techniques in cloud computing. *Journal of Network and Computer Applications*, 2019.
- [18] Felipe S Dantas Silva, Marcilio OO Lemos, Alisson Medeiros, Augusto Venâncio Neto, Rafael Pasquini, David Moura, Christian Rothenberg, Lefteris Mamatas, Sand Luz Correa, Kleber Vieira Cardoso, et al. Necos project: Towards lightweight slicing of cloud federated infrastructures. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 406–414. IEEE, 2018.
- [19] Mostafa Ghobaei-Arani, Alireza Souri, and Ali A Rahmanian. Resource management approaches in fog computing: a comprehensive review. *Journal of Grid Computing*, pages 1–42, 2019.
- [20] Quan Yuan, Xinsheng Ji, Hongbo Tang, and Wei You. Toward latency-optimal placement and autoscaling of monitoring functions in mec. *IEEE Access*, 8:41649–41658, 2020.
- [21] Nan Wang, Blesson Varghese, Michail Matthaiou, and Dimitrios S Nikolopoulos. Enorm: A framework for edge node resource management. *IEEE transactions on services computing*, 2017.
- [22] Rodrigo da Rosa Righi, Leandro Andrioli, Vinicius Facco Rodrigues, Cristiano André da Costa, Antonio Marcos Alberti, and Dhananjay Singh. Elastic-ran: an adaptable multi-level elasticity model for cloud radio access networks. *Computer Communications*, 142:34–47, 2019.
- [23] Chunlin Li, Hezhi Sun, Yi Chen, and Youlong Luo. Edge cloud resource expansion and shrinkage based on workload for minimizing the cost. *Future Generation Computer Systems*, 101:327–340, 2019.
- [24] Alexandru-Florian Antonescu and Torsten Braun. Simulation of sla-based vm-scaling algorithms for cloud-distributed applications. *Future Generation Computer Systems*, 54:260–273, 2016.
- [25] Ranesh Kumar Naha, Saurabh Garg, Andrew Chan, and Sudheer Kumar Battula. Deadline-based dynamic resource allocation and provisioning algorithms in fog-cloud environment. *Future Generation Computer Systems*, 104:131–141, 2020.
- [26] Chunlin Li, Chengyi Wang, and Youlong Luo. An efficient scheduling optimization strategy for improving consistency maintenance in edge cloud environment. *The Journal of Supercomputing*, pages 1–28, 2020.
- [27] Gabriele Castellano, Flavio Esposito, and Fulvio Risso. A distributed orchestration algorithm for edge computing resources with guarantees. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 2548–2556. IEEE, 2019.
- [28] Argyrios G Tasiopoulos, Onur Ascigil, Ioannis Psaras, and George Pavlou. Edge-map: Auction markets for edge resource provisioning. In *2018 IEEE 19th International Symposium on "A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*, pages 14–22. IEEE, 2018.
- [29] Jingjing Guo, Chunlin Li, Yi Chen, and Youlong Luo. On-demand resource provision based on load estimation and service expenditure in edge cloud environment. *Journal of Network and Computer Applications*, 151:102506, 2020.
- [30] Ioannis Sarrigiannis, Kostas Ramantas, Elli Kartsakli, Prodromos Vasileios Mekikis, Angelos Antonopoulos, and Christos Verikoukis. Online vnf lifecycle management in an mec-enabled 5g iot architecture. *IEEE Internet of Things Journal*, 7(5):4183–4194, 2019.
- [31] Nabeel Akhtar, Ibrahim Matta, Ali Raza, Leonardo Goratti, Torsten Braun, and Flavio Esposito. Managing chains of application functions over multi-technology edge networks. *IEEE Transactions on Network*

*and Service Management*, 2021.

- [32] Hanieh Alipour, Yan Liu, and Abdelwahab Hamou-Lhadj. Analyzing auto-scaling issues in cloud environments. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, CASCON '14, page 75–89, USA, 2014. IBM Corp.
- [33] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.
- [34] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
- [35] Alisson Medeiros. REACT Prototype, 2020. URL <https://bitbucket.org/alissonmedeiros/elasticity/src/master/>.



**Alisson Medeiros** is a Ph.D. candidate at the Communication and Distributed Systems (CDS) group, Institute of Computer Science, University of Bern, Switzerland. He received his MSc from the Federal University of Rio Grande do Norte (UFRN), Brazil, and got his BSc in computer science at the State University of Paraíba, Brazil. His research interest includes resource management and orchestration of Cloud/Fog/Edge Computing systems.



**Antonio Di Maio** is a postdoctoral researcher in mobile networks with the Communication and Distributed Systems (CDS) group at the University of Bern, Switzerland. He obtained his PhD degree in Computer Engineering from the University of Luxembourg in 2020, with a thesis on routing and content dissemination in software-defined vehicular networks. His current research interests fall within the areas of network modeling, scheduling, routing, and channel access.



**Torsten Braun** is currently director at the Institute of Computer Science, University of Bern, where he has been a full professor since 1998. He got the Ph.D. degree from University of Karlsruhe (Germany) in 1993. From 1994 to 1995, he was a guest scientist at INRIA Sophia-Antipolis (France). From 1995 to 1997, he worked at the IBM European Networking Centre Heidelberg (Germany) as a project leader and senior consultant. He has been a vice president of the SWITCH (Swiss Research and Education Network Provider) Foundation from 2011 to 2019. He has been a Director of the Institute of Computer Science and Applied Mathematics at University of Bern between 2007 and 2011, and since 2019.



**Augusto Neto** is Associate Professor at the Informatics and Applied Mathematics Department (DIMAp) of the Federal University of Rio Grande do Norte (UFRN), member of the Instituto de Telecomunicações (Aveiro, Portugal), researcher of the National Council for Technological and Scientific Development (CNPq), and leader of the Research Group on Future Internet Service & Applications (REGINA), working mainly in the fields of Computer Networks and Distributed Systems. He got his Ph.D. at the University of Coimbra (2008), and coordinates/participates in research projects funded by national and international development agencies.